Automatic Constrained Rewriting Induction Towards Verifying Procedural Programs *

Cynthia Kop[1] and Naoki Nishida[2]

[1] Institute of Computer Science, University of Innsbruck
Cynthia.Kop@uibk.ac.at
[2] Graduate School of Information Science, Nagoya University
nishida@is.nagoya-u.ac.jp

Abstract. This paper aims at developing a verification method for procedural programs via a transformation into logically constrained term rewriting systems (LCTRSs). To this end, we adapt existing rewriting induction methods to LCTRSs and propose a simple yet effective method to generalize equations. We show that we can handle realistic functions, involving, e.g., integers and arrays. An implementation is provided.

1 Introduction

A problem familiar to many computer science lecturers, is the marking of student programming assignments. This can be large time drain, as it typically involves checking dozens (or hundreds!) of unnecessarily complicated programs at once.

An obvious solution is automatic testing. For example, one might run assignments on a fixed set of input files; this quickly weeds out incorrect solutions, but has a high risk of false positives. Alternatively (or in addition), we can try to automatically prove correctness. Several methods for this have been investigated (see e.g. [9]). However, most of them require expert knowledge to use, like *assertions* in the code to trace relevant properties; this is not useful in our setting.

An interesting alternative is *inductive theorem proving*, which is well investigated in the field of functional programming (see, e.g., [2]). For a functional program f to be checked against a specification f_{spec}, it suffices if $f(\overrightarrow{x}) \approx f_{spec}(\overrightarrow{x})$ is an *inductive theorem* of the combined system of f and f_{spec}. For this initial setting, no expert knowledge is needed, only the definitions of f and f_{spec}.

Recently, analyses of procedural programs (in C, Java Bytecode, etc.) via transformations into term rewriting systems have been investigated [4,6,8,17]. In particular, *constrained rewriting systems* are popular for these transformations, since logical constraints used for modeling the control flow can be separated from terms expressing intermediate states [4,6,8,16,20]. To capture the existing approaches for constrained rewriting in one setting, the framework of *logically constrained term rewriting systems* (LCTRS) has been proposed [13].

* This research is supported by the Austrian Science Fund (FWF) international project I963, the Japan Society for the Promotion of Science (JSPS) and *Nagoya University's Graduate Program for Real-World Data Circulation Leaders* from *MEXT*, Japan.

In this paper, we develop a verification method for LCTRSs, designed in particular for LCTRSs obtained from procedural programs. We use rewriting induction [18], one of the well-investigated methods for inductive theorem proving, together with a generalization technique that works particularly well for transformed iterative functions. Although our examples focus on integers and static integer arrays, the results can be used with various theories.

Of course, verification also has applications outside the academic world. Although we initially focus on typical homework assignments (small programs, which require only limited language features), we hope to additionally lay a basis for more extensive program analysis using constrained term rewriting systems.

In this paper, we first recall the LCTRS formalism from [13] (Section 2), and sketch a way to translate procedural programs to LCTRSs (Section 3). Then we adapt existing rewriting induction methods for earlier notions of constrained rewriting [5,20] to LCTRSs (Section 4), which is strengthened with a dedicated generalization technique (Section 5). Finally, we briefly discuss implementation ideas (Section 6), give a comparison with related work (Section 7) and conclude.

An extended version of this paper, including all proofs, is available in [14].

2 Preliminaries

In this section, we briefly recall *Logically Constrained Term Rewriting Systems (LCTRSs)*, following the definitions in [13].

Many-sorted Terms. We assume given a set $\mathcal{S}$ of *sorts* and an infinite set $\mathcal{V}$ of *variables*, each variable equipped with a sort. A *signature* Σ is a set of *function symbols* f, disjoint from $\mathcal{V}$, each symbol equipped with a *sort declaration* $[\iota_1 \times \cdots \times \iota_n] \Rightarrow \kappa$, with all ι_i and κ sorts. The set $\mathcal{T}erms(\Sigma, \mathcal{V})$ of *terms* over Σ and $\mathcal{V}$, contains any expression s such that $\vdash s : \iota$ can be derived for some sort ι, using:

$$\frac{}{\vdash x : \iota}\ (x : \iota \in \mathcal{V}) \qquad \frac{\vdash s_1 : \iota_1 \quad \dots \quad \vdash s_n : \iota_n}{\vdash f(s_1, \dots, s_n) : \kappa}\ (f : [\iota_1 \times \cdots \times \iota_n] \Rightarrow \kappa \in \Sigma)$$

Fixing Σ and $\mathcal{V}$, every term has a unique sort ι such that $\vdash s : \iota$; we say that ι is the sort of s. Let $Var(s)$ be the set of variables occurring in s. A term s is *ground* if $Var(s) = \emptyset$. A *substitution* is a sort-preserving mapping $[x_1 := s_1, \dots, x_k := s_k]$ from variables to terms; $s\gamma$ denotes s with occurrences of any x_i replaced by s_i.

Given a term s, a *position* in s is a sequence p of integers such that $s_{|p}$ is defined, where $s_{|\epsilon} = s$ and $f(s_1, \dots, s_n)_{|i \cdot p} = (s_i)_{|p}$. We say that $s_{|p}$ is a *subterm* of s. If $\vdash s_{|p} : \iota$ and $\vdash t : \iota$, then $s[t]_p$ denotes s with the subterm at position p replaced by t. A *context* C is a term containing one or more typed *holes* $\Box_i : \iota_i$. If $s_1 : \iota_i, \dots, s_n : \iota_n$, we define $C[s_1, \dots, s_n]$ as C with each $\Box_i$ replaced by s_i.

Logical Terms. We fix a signature $\Sigma = \Sigma_{terms} \cup \Sigma_{theory}$ (with possible overlap, as discussed below). The sorts occurring in Σ_{theory} are called *theory sorts*, and the symbols *theory symbols*. We assume given a mapping $\mathcal{I}$ which assigns to each theory sort ι a set $\mathcal{I}_\iota$, and a mapping $\mathcal{J}$ which maps each $f : [\iota_1 \times \cdots \times \iota_n] \Rightarrow \kappa \in \Sigma_{theory}$ to a function $\mathcal{J}_f$ in $\mathcal{I}_{\iota_1} \times \cdots \times \mathcal{I}_{\iota_n} \Longrightarrow \mathcal{I}_\kappa$. For all theory sorts ι we also fix a

set $\mathcal{V}al_\iota \subseteq \Sigma_{theory}$ of *values*: function symbols $a : [] \Rightarrow \iota$, where $\mathcal{J}$ gives a bijective mapping from $\mathcal{V}al_\iota$ to $\mathcal{I}_\iota$. We require that $\Sigma_{terms} \cap \Sigma_{theory} \subseteq \mathcal{V}al = \bigcup_\iota \mathcal{V}al_\iota$.

A term in $\mathcal{T}erms(\Sigma_{theory}, \mathcal{V})$ is called a *logical term*. For ground logical terms, let $[\![f(s_1, \ldots, s_n)]\!] := \mathcal{J}_f([\![s_1]\!], \ldots, [\![s_n]\!])$. Every ground logical term s corresponds to a unique value c such that $[\![s]\!] = [\![c]\!]$; we say that c is the value of s. A *constraint* is a logical term φ of some sort bool with $\mathcal{I}_{\mathsf{bool}} = \mathbb{B} = \{\top, \bot\}$, the set of *booleans*. We say φ is *valid* if $[\![\varphi\gamma]\!] = \top$ for *all* substitutions γ which map $Var(\varphi)$ to values, and *satisfiable* if $[\![\varphi\gamma]\!] = \top$ for *some* substitution γ which maps $Var(\varphi)$ to values. A substitution γ *respects* φ if $\gamma(x)$ is a value for all $x \in Var(\varphi)$ and $[\![\varphi\gamma]\!] = \top$.

Formally, terms in $\mathcal{T}erms(\Sigma_{terms}, \mathcal{V})$ have no special function, but we see them as the primary objects of the term rewriting system: a reduction would typically begin and end with such terms, with elements of $\Sigma_{theory} \setminus \mathcal{V}al$ (also called *calculation symbols*) only used in intermediate terms.

We typically choose a theory signature with $\Sigma_{theory} \supseteq \Sigma^{core}_{theory}$, where Σ^{core}_{theory} contains the core theory symbols: $\mathsf{true}, \mathsf{false} : \mathsf{bool}, \wedge, \vee, \Rightarrow\colon [\mathsf{bool} \times \mathsf{bool}] \Rightarrow \mathsf{bool}$, $\neg\colon [\mathsf{bool}] \Rightarrow \mathsf{bool}$, and, for all sorts ι, symbols $=_\iota, \neq_\iota\colon [\iota \times \iota] \Rightarrow \mathsf{bool}$, and an evaluation function $\mathcal{J}$ that interprets these symbols as expected. We omit the sort subscripts from $=$ and $\neq$ when they can be derived from context.

The standard integer signature Σ^{int}_{theory} is $\Sigma^{core}_{theory} \cup \{+, -, *, \mathsf{exp}, \mathsf{div}, \mathsf{mod} : [\mathsf{int} \times \mathsf{int}] \Rightarrow \mathsf{int}; \leq, <\colon [\mathsf{int} \times \mathsf{int}] \Rightarrow \mathsf{bool}\} \cup \{\mathsf{n} : \mathsf{int} \mid n \in \mathbb{Z}\}$. Here, values are true, false and n for all $n \in \mathbb{Z}$. We let $\mathcal{J}$ be defined in the natural way, but (since all $\mathcal{J}_f$ must be total) $\mathcal{J}_{\mathsf{div}}(n, 0) = \mathcal{J}_{\mathsf{mod}}(n, 0) = \mathcal{J}_{\mathsf{exp}}(n, k) = 0$ for all n and all $k < 0$. However, when constructing LCTRSs, we normally avoid such calls.

Rules and Rewriting. A *rule* is a triple $\ell \to r\ [\varphi]$ where ℓ and r are terms of the same sort and φ is a constraint. Here, ℓ is not a logical term (so also not a variable, as $\mathcal{V} \subseteq \mathcal{T}erms(\Sigma_{theory}, \mathcal{V})$). If $\varphi = \mathsf{true}$ with $\mathcal{J}(\mathsf{true}) = \top$, the rule is usually just denoted $\ell \to r$. We define $LVar(\ell \to r\ [\varphi])$ as $Var(\varphi) \cup (Var(r) \setminus Var(\ell))$. A substitution γ *respects* $\ell \to r\ [\varphi]$ if $\gamma(x)$ is a value for all $x \in LVar(\ell \to r\ [\varphi])$, and $\varphi\gamma$ is valid. Note that it is allowed that $Var(r) \not\subseteq Var(\ell)$, but fresh variables in the right-hand side may only be instantiated with *values*. This is done to model user input or random choice, both of which would typically produce a value. Variables on the left do not need to be instantiated with values (unless they also occur in the constraint); this is needed for instance for lazy evaluation.

We assume given a set of rules $\mathcal{R}$, and let $\mathcal{R}_{\mathtt{calc}}$ be the set $\{f(x_1, \ldots, x_n) \to y\ [y = f(\overrightarrow{x})] \mid f : [\iota_1 \times \cdots \times \iota_n] \Rightarrow \kappa \in \Sigma_{theory} \setminus \mathcal{V}al\}$ (writing $\overrightarrow{x}$ for $x_1, \ldots, x_n$). The *rewrite relation* $\to_{\mathcal{R}}$ is a binary relation on terms, defined by:

$$C[\ell\gamma] \to_{\mathcal{R}} C[r\gamma] \text{ if } \ell \to r\ [\varphi] \in \mathcal{R} \cup \mathcal{R}_{\mathtt{calc}} \text{ and } \gamma \text{ respects } \ell \to r\ [\varphi]$$

We say the reduction occurs at position p if $C = C[\Box]_p$. Let $s \leftrightarrow_{\mathcal{R}} t$ if $s \to_{\mathcal{R}} t$ or $t \to_{\mathcal{R}} s$. A reduction step with $\mathcal{R}_{\mathtt{calc}}$ is called a *calculation*. A term is in *normal form* if it cannot be reduced with $\to_{\mathcal{R}}$. If $f(\ell_1, \ldots, \ell_n) \to r\ [\varphi] \in \mathcal{R}$ we call f a *defined symbol*; non-defined elements of Σ_{terms} and all values are *constructors*. Let $\mathcal{C}ons$ be the set of all constructors. A *logically constrained term rewriting system* (LCTRS) is the abstract rewriting system $(\mathcal{T}erms(\Sigma, \mathcal{V}), \to_{\mathcal{R}})$, usually given by supplying Σ, $\mathcal{R}$, and maybe $\mathcal{I}$ and $\mathcal{J}$ if these are not clear from context.

Example 1. To implement an LCTRS calculating the *factorial* function, we let $\mathcal{I}_{\mathsf{int}} = \mathbb{Z}$, $\mathcal{I}_{\mathsf{bool}} = \mathbb{B}$, $\Sigma_{theory} = \Sigma^{int}_{theory}$, $\mathcal{J}$ defined as discussed above, and:

$$\Sigma_{terms} = \{\ \mathsf{fact} : [\mathsf{int}] \Rightarrow \mathsf{int}\} \cup \{\mathsf{n} : \mathsf{int} \mid n \in \mathbb{Z}\ \}$$
$$\mathcal{R}_{fact} = \{\ \mathsf{fact}(x) \to 1\ [x \leq 0]\ \ ,\ \ \mathsf{fact}(x) \to x * \mathsf{fact}(x-1)\ [\neg(x \leq 0)]\ \}$$

Using infix notation, examples of logical terms are $5 + 9$ and $0 = 0 + -1$ and $x + 3 \geq y + -42$; the latter two are constraints. We can reduce $5 + 9$ to 14 with a calculation (using $x + y \to z\ [z = x + y]$), and $\mathsf{fact}(3)$ reduces in ten steps to 6.

Example 2. To implement an LCTRS calculating the sum of elements in an array, let $\mathcal{I}_{\mathsf{bool}} = \mathbb{B}$, $\mathcal{I}_{\mathsf{int}} = \mathbb{Z}$ and $\mathcal{I}_{\mathsf{array(int)}} = \mathbb{Z}^*$, so $\mathsf{array(int)}$ is mapped to finite-length integer sequences. Let $\Sigma_{theory} = \Sigma^{int}_{theory} \cup \{\mathsf{size} : [\mathsf{array(int)}] \Rightarrow \mathsf{int},\ \mathsf{select} : [\mathsf{array(int)} \times \mathsf{int}] \Rightarrow \mathsf{int}\} \cup \{\mathsf{a} \mid a \in \mathbb{Z}^*\}$. (So we do not encode arrays as lists: every array a corresponds to a unique symbol a.) The interpretation function $\mathcal{J}$ behaves on Σ^{int}_{theory} as usual and has $\mathcal{J}_{\mathsf{size}}(a) = k$ when $a = \langle n_0, \ldots, n_{k-1}\rangle$, and $\mathcal{J}_{\mathsf{select}}(a, i) = n_i$ if $a = \langle n_0, \ldots, n_{k-1}\rangle$ with $0 \leq i < k$, otherwise 0. In addition:

$$\Sigma_{terms} = \{\ \mathsf{sum} : [\mathsf{array(int)}] \Rightarrow \mathsf{int},\ \mathsf{sum1} : [\mathsf{array(int)} \times \mathsf{int}] \Rightarrow \mathsf{int}\ \} \cup \{\ \mathsf{n} : \mathsf{int} \mid n \in \mathbb{Z}\ \} \cup \{\ \mathsf{a} \mid a \in \mathbb{Z}^*\ \}$$
$$\mathcal{R}_{\mathsf{sum}} = \left\{ \begin{array}{ll} \mathsf{sum}(x) \to \mathsf{sum1}(x, \mathsf{size}(x) - 1) & \\ \mathsf{sum1}(x, k) \to \mathsf{select}(x, k) + \mathsf{sum1}(x, k-1) & [k \geq 0] \\ \mathsf{sum1}(x, k) \to 0 & [k < 0] \end{array} \right\}$$

Note the special role of *values*, which are new in LCTRSs compared to older styles of constrained rewriting. They are the representatives of the underlying theory. All values are constants (constructor symbols $v()$ which do not take arguments), even if they represent complex structures, as seen in Example 2. However, not all constants are values. Because, unlike traditional TRSs and e.g. [6,8], values are not term-generated. we can easily have uncountably many of them (for example an LCTRS over the real number field $\mathbb{R}$), and do not have to match modulo theories (for example equating $0 + (x + y)$ with $y + x$).

Quantification. The definition of LCTRSs does *not* permit quantifiers. In for instance an LCTRS over integers and arrays, we cannot specify a rule $\mathsf{extend}(arr, x) \to \mathsf{addtoend}(x, arr)\ [\forall y \in \{0, \ldots, \mathsf{size}(arr) - 1\} : x \neq \mathsf{select}(arr, y)]$ (where $\mathsf{addtoend} : [\mathsf{int} \times \mathsf{array(int)}] \Rightarrow \mathsf{array(int)} \in \Sigma_{theory}$ and extend is a defined symbol).

However, one of the key features of LCTRSs is that theory symbols, including predicates, are not confined to a fixed list. Therefore, what we *can* do when defining an LCTRS, is to add a new symbol to Σ_{theory} (and $\mathcal{J}$). For the extend rule, we could for instance introduce a symbol $\mathsf{notin} : [\mathsf{int} \times \mathsf{array(int)}] \Rightarrow \mathsf{bool}$ with $\mathcal{J}_{\mathsf{notin}}(u, \langle a_0, \ldots, a_{n-1}\rangle) = \top$ if for all i: $u \neq a_i$, and replace the constraint by $\mathsf{notin}(x, arr)$. This generates the same reduction relation as the original rule.

Thus, we can permit quantifiers in the constraints of rules, as intuitive notation for fresh predicates. However, as the reduction relation $\to_{\mathcal{R}}$ is only decidable if all $\mathcal{J}_f$ are, an *unbounded* quantification would likely not be useful in practice.

Differences to [13]. In [13], where LCTRSs are first defined, we assume that $\mathcal{V}$ contains unsorted variables, and use a separate *variable environment* for typing terms. Also, $\to_{\mathcal{R}}$ is there defined as the union of $\to_{\mathtt{rule}}$ (using rules in $\mathcal{R}$) and $\to_{\mathtt{calc}}$ (using calculations). These changes give equivalent results, but the current definitions cause a bit less bookkeeping.

A non-equivalent change is the requirement on rules: in [13] left-hand sides must have a root symbol in $\Sigma_{terms} \setminus \Sigma_{theory}$. We follow [12] in weakening this.

2.1 Rewriting Constrained Terms

In LCTRSs, the objects of study are *terms*, with $\to_{\mathcal{R}}$ defining the relation between them. However, for analysis it is often useful to consider *constrained terms*: pairs $s\,[\varphi]$ of a term s and a constraint φ. A constrained term $s\,[\varphi]$ represents all terms $s\gamma$ where γ respects φ, and can be used to reason about such terms.

Different constrained terms might represent the same terms; for example $f(\mathsf{0})\,[\mathsf{true}]$ and $f(x)\,[x = \mathsf{0}]$, or $g(x, y)\,[x > y]$ and $g(z, u)\,[u \leq z - \mathsf{1}]$. We consider these terms *equivalent*. Formally, $s\,[\varphi] \sim t\,[\psi]$ if for all substitutions γ which respect φ there is a substitution δ which respects ψ such that $s\gamma = t\delta$, and vice versa. Note that $s\,[\varphi] \sim s\,[\psi]$ if and only if $\forall\overrightarrow{x}[\exists\overrightarrow{y}[\varphi] \leftrightarrow \exists\overrightarrow{z}[\psi]]$ holds, where $Var(s) = \{\overrightarrow{x}\}$, $Var(\varphi) \setminus Var(s) = \{\overrightarrow{y}\}$ and $Var(\psi) \setminus Var(s) = \{\overrightarrow{z}\}$.

For a rule $\rho := \ell \to r\ [\psi] \in \mathcal{R} \cup \mathcal{R}_{\mathtt{calc}}$ and position q, we let $s\,[\varphi] \to_{\rho,q} t\,[\varphi]$ if $s_{|q} = \ell\gamma$ and $t = s[r\gamma]_q$ for some substitution γ with $\gamma(x)$ a variable in $Var(\varphi)$ or value for all $x \in LVar(\rho)$ and $\varphi \Rightarrow (\psi\gamma)$ valid. Let $s\,[\varphi] \to_{\mathtt{base}} t\,[\varphi]$ if $s\,[\varphi] \to_{\rho,q} t\,[\varphi]$ for some ρ, q. The relation $\to_{\mathcal{R}}$ on constrained terms is: $\sim \cdot \to_{\mathtt{base}} \cdot \sim$. We say $s\,[\varphi] \to_{\mathcal{R}} t\,[\psi]$ at position q by rule ρ if $s\,[\varphi] \sim \cdot \to_{\rho,q} \cdot \sim t\,[\psi]$.

Example 3. In the factorial LCTRS from Example 1, we have that $\mathsf{fact}(x)\,[x > \mathsf{3}] \to_{\mathcal{R}} x * \mathsf{fact}(x - \mathsf{1})\,[x > \mathsf{3}]$. This constrained term can be further reduced using the calculation rule $x - y \to z\ [z = x - y]$, but here we must use the $\sim$ relation, as follows: $x * \mathsf{fact}(x - \mathsf{1})\,[x > \mathsf{3}] \sim x * \mathsf{fact}(x - \mathsf{1})\,[x > \mathsf{3} \wedge z = x - \mathsf{1}] \to_{\mathtt{base}} x * \mathsf{fact}(z)\,[x > \mathsf{3} \wedge z = x - \mathsf{1}]$, as $\forall x[x > \mathsf{3} \leftrightarrow \exists z[x > \mathsf{3} \wedge z = x - \mathsf{1}]]$.

Example 4. The relation $\sim$ allows us to reformulate the constraint both before and after a reduction, which is particularly useful for *irregular* rules, where the constraint contains variables not occurring in the left-hand side. The calculation rules are a particular example of such rules, as we saw in Example 3. For a different example, with the rule $f(x) \to g(y)\ [y > x]$, we have: $f(x)\,[x > \mathsf{3}] \sim f(x)\,[x > \mathsf{3} \wedge y > x] \to_{\mathtt{base}} g(y)\,[x > \mathsf{3} \wedge y > x] \sim g(y)\,[y > \mathsf{4}]$. Similarly, $f(x - \mathsf{1})\,[x > \mathsf{0}]$ reduces with a calculation to $f(y)\,[y \geq \mathsf{0}]$. We do *not* have that $f(x)\ [\mathsf{true}] \to_{\mathcal{R}} g(x + \mathsf{1})\ [\mathsf{true}]$, as $x + \mathsf{1}$ cannot be instantiated to a value.

Example 5. A constrained term does not always need to be reduced in the most general way. With the rule $f(x) \to g(y)\ [y > x]$, we have $f(\mathsf{0})\ [\mathsf{true}] \sim f(\mathsf{0})\ [y > \mathsf{0}] \to_{\mathtt{base}} g(y)\ [y > \mathsf{0}]$, but we also have $f(\mathsf{0})\ [\mathsf{true}] \sim f(\mathsf{0})\ [\mathsf{1} > \mathsf{0}] \to_{\mathtt{base}} g(\mathsf{1})$.

As intended, constrained reductions give information about usual reductions:

Theorem 6 ([13]). *If $s\,[\varphi] \to_{\mathcal{R}} t\,[\psi]$, then for all substitutions γ which respect φ there is a substitution δ which respects ψ such that $s\gamma \to_{\mathcal{R}}^{+} t\delta$.*

3 Transforming Imperative Programs into LCTRSs

Transformations of imperative programs into integer rewriting systems are investigated in e.g. [4,6,8]. These papers use different variations of constrained rewriting, but the proposed transformations are easily adapted to produce LCTRSs that operate on integers, i.e., use Σ^{int}_{theory}. What is more, we can extend the ideas to also handle advanced programming structures, like function calls and arrays.

Following the ideas of [4,6,8], we transform each function f separately. Let $\overrightarrow{v}$ be the vector of all parameters and local variables in f (we disallow global variables for now). For all basic blocks in the function (i.e., straight-line code segments), we introduce a new function symbol u_i. A transition from block i to block j is encoded as a rule $\mathsf{u}_i(\overrightarrow{v}) \to \mathsf{u}_j(\overrightarrow{r})\ [\varphi]$, with assignments reflected by argument updates in the right-hand side, and conditions by the constraint. Return statements `return e` are encoded by reducing to $\mathsf{return}_\mathsf{f}(e)$, where $\mathsf{return}_\mathsf{f}$ is a new constructor.

Finally, the generated LCTRS is optimized to make it more amenable to analysis: we combine rules whose root symbols occur only once in left-hand sides [6], remove unused parameters (in particular, variables not in scope at a given location), and, if appropriate, simplify the constraint (e.g. by removing duplicate clauses or replacing a term like $\neg(x > y)$ by $y \geq x$.

Example 7. Consider the following small C-function fact, calculating the factorial function from Example 1. Here, $\overrightarrow{v}$ is $\langle x, i, z\rangle$. There are three basic blocks: u_1 (the initialization of the local variables, which includes both `int z = 1` and `int i = 1`), u_2 (the loop body), and u_3 (the block containing the return-statement).

```
int fact(int x) {
  int z = 1;
  for(int i = 1; i <= x; i++)
    z *= i;
  return z;
}
```

We obtain the following initial LCTRS (left) and simplification (right):

$$
\begin{array}{l}
\mathsf{fact}(x) \to \mathsf{u}_1(x,i,z) \\
\mathsf{u}_1(x,i,z) \to \mathsf{u}_2(x,1,1) \\
\mathsf{u}_2(x,i,z) \to \mathsf{u}_2(x,i+1,z*i)\ [i \leq x] \\
\mathsf{u}_2(x,i,z) \to \mathsf{u}_3(x,i,z)\ \ [\neg(i \leq x)] \\
\mathsf{u}_3(x,i,z) \to \mathsf{return}_{\mathsf{fact}}(z)
\end{array}
\qquad
\begin{array}{l}
\mathsf{fact}(x) \to \mathsf{u}_2(x,1,1) \\
\mathsf{u}_2(x,i,z) \to \mathsf{u}_2(x,i+1,z*i)\ [i \leq x] \\
\mathsf{u}_2(x,i,z) \to \mathsf{return}_{\mathsf{fact}}(z)\ \ [i > x]
\end{array}
$$

Note that there is nothing special about the integers; the definition of LCTRSs allows values from all kinds of underlying domains. So, with a suitable theory signature, we could also handle e.g. doubles, encoding them as real numbers. *Pointers* are more difficult to handle, but *static arrays* are not so problematic. Consider for instance the following two implementations of the same assignment: *given an integer array and its length, return the sum of the array's elements.*

```
int sum1(int arr[],int n){
  int ret=0;
  for(int i=0;i<n;i++)
    ret+=arr[i];
  return ret;
}
```

```
int sum2(int *arr, int k) {
  if (k <= 0) return 0;
  return arr[k-1] +
         sum2(arr, k-1);
}
```

To encode these functions, we use Σ_{theory} as in Example 2. To handle illegal program behavior, we reduce to an additional $\mathsf{error_f}$ constructor in cases when we index an array out of bounds. To handle function calls (as in $\mathsf{sum2}$), we execute the call in a separate parameter, and then examine the result. These ideas result in the following simplified translations (using the same return and error symbols in both cases, because we want to be able to compare the resulting functions):

$$
\begin{array}{lll}
(1) & \mathsf{sum1}(arr, n) \to \mathsf{u}(arr, n, 0, 0) & \\
(2) & \mathsf{u}(arr, n, ret, i) \to \mathsf{error} & [i < n \wedge (i < 0 \vee i \geq \mathsf{size}(arr))] \\
(3) & \mathsf{u}(arr, n, ret, i) \to \mathsf{u}(arr, n, ret + \mathsf{select}(arr, i), i + 1) & [i < n \wedge 0 \leq i < \mathsf{size}(arr)] \\
(4) & \mathsf{u}(arr, n, ret, i) \to \mathsf{return}(ret) & [i \geq n] \\
(5) & \mathsf{sum2}(arr, k) \to \mathsf{return}(0) & [k \leq 0] \\
(6) & \mathsf{sum2}(arr, k) \to \mathsf{error} & [k - 1 \geq \mathsf{size}(arr)] \\
(7) & \mathsf{sum2}(arr, k) \to \mathsf{w}(\mathsf{select}(arr, k - 1), \mathsf{sum2}(arr, k - 1)) & [0 \leq k - 1 < \mathsf{size}(arr)] \\
(8) & \mathsf{w}(n, \mathsf{error}) \to \mathsf{error} & \\
(9) & \mathsf{w}(n, \mathsf{return}(r)) \to \mathsf{return}(n + r) &
\end{array}
$$

Here, a constraint $x \leq y < b$ should be read as: $x \leq y \wedge y < b$. Note that $\mathsf{sum2}$ differs from the system in Example 2 only by adding error-handling.

In general, we can encode arrays of any data type, including arrays of arrays, by defining $\mathcal{I}_{\mathsf{array}(\iota)} = \mathcal{I}_\iota^*$ for any ι with $\mathcal{J}_\iota \neq \emptyset$ (we need some default value $\mathsf{0}_\iota \in \mathcal{V}al_\iota$ for out-of-bound selects). We can also handle array updates: let $\mathsf{store} : [\mathsf{array}(\iota) \times \mathsf{int} \times \iota] \Rightarrow \mathsf{array}(\iota)$, and $\mathcal{J}_{\mathsf{store}}(\langle a_0, \ldots, a_{n-1}\rangle, k, v) = \langle a_0, \ldots, a_{k-1}, v, a_{k+1}, \ldots, a_{n-1}\rangle$ if $0 \leq k < n$ and $\langle \overrightarrow{a} \rangle$ otherwise. To reflect side effects, we include updated array parameters in the return value.

Example 8. The function `void empty(char arr[]) { arr[0] = '\0'; }` is translated to the following LCTRS:

$$
\begin{array}{ll}
\mathsf{empty}(arr) \to \mathsf{error}_{\mathsf{empty}} & [0 \geq \mathsf{size}(arr)] \\
\mathsf{empty}(arr) \to \mathsf{return}(\mathsf{store}(arr, 0, 0)) & [0 < \mathsf{size}(arr)]
\end{array}
$$

A more extensive discussion of this translation, including global variables, integer overflow and dynamic pointers, is available online in [14, Section 3].

4 Rewriting Induction for LCTRSs

In this section, we adapt the inference rules from [18,5,20] to inductive theorem proving with LCTRSs. This provides the core theory to use rewriting induction, which will be strengthened with a lemma generalization technique in Section 5.

We start by listing some restrictions we need to impose on LCTRSs for the method to work (Section 4.1). Then, we provide the theory for the technique (Section 4.2), making several changes compared to [18,5,20] to handle the new formalism. We complete with two illustrative examples (Section 4.3).

4.1 Restrictions

In order for rewriting induction to be successful, we need to impose certain restrictions. We limit interest to LCTRSs which satisfy the following properties:

1. all core theory symbols ($\wedge, \vee, \Rightarrow, \neg$ and each $=_\iota, \neq_\iota$) are present in Σ_{theory};
2. the LCTRS is terminating, so there is no infinite reduction $s_1 \to_{\mathcal{R}} s_2 \to_{\mathcal{R}} \ldots$;
3. the system is *quasi-reductive*, i.e., for every term s either $s \in \mathcal{T}erms(\mathcal{C}ons, \emptyset)$ (we say s is a *ground constructor term*), or there is some t such that $s \to_{\mathcal{R}} t$;[3]
4. there are ground terms of every sort occurring in Σ.

Property 1 is just the standard assumption we saw in Section 2. We will need these symbols, for instance, to add new information to a constraint. Termination (property 2) is crucial in the inductive derivation, as the method uses induction on terms, oriented with an extension of $\to_{\mathcal{R}}$. Property 3 which, together with termination, provides *sufficient completeness*, makes it possible to do an exhaustive case analysis on the rules applicable to an equation. It also allows us to assume that variables are always instantiated by ground constructor terms. The last property is natural, since the method considers *ground* terms; function symbols which cannot be assigned ground arguments can simply be omitted.

Methods to prove quasi-reductivity and termination have been published for different styles of constrained rewriting; see e.g. [5] for quasi-reductivity and [7,19] for termination. These methods are easily adapted to LCTRSs: see [14, Appendix A] for quasi-reductivity and [12] for termination. The LCTRSs obtained from procedural programs following Section 3 are always quasi-reductive.

4.2 Rewriting Induction

We now introduce the notions of *constrained equations* and *inductive theorems*.

Definition 9. *A* (constrained) equation *is a triple $s \approx t\ [\varphi]$ with s and t terms and φ a constraint. Let $s \simeq t\ [\varphi]$ denote either $s \approx t\ [\varphi]$ or $t \approx s\ [\varphi]$. A substitution γ* respects *$s \approx t\ [\varphi]$ if γ respects φ and $Var(s) \cup Var(t) \subseteq Dom(\gamma)$. We say γ is a* ground constructor substitution *if all $\gamma(x)$ are ground constructor terms.*

An equation $s \approx t\ [\varphi]$ is an inductive theorem *of an LCTRS $\mathcal{R}$ if $s\gamma \leftrightarrow^*_{\mathcal{R}} t\gamma$ for any ground constructor substitution γ that respects this equation.*

Intuitively, if an equation $f(\overrightarrow{x}) \approx g(\overrightarrow{x})\ [\varphi]$ is an inductive theorem, then f and g define the same function (conditional on φ, and assuming confluence).

To prove that an equation is an inductive theorem, we will consider five inference rules, originating in [18,5,20]. These rules modify a *proof state*: a pair $(\mathcal{E}, \mathcal{H})$ where $\mathcal{E}$ is a set of equations and $\mathcal{H}$ a set of constrained rewrite rules with $\to_{\mathcal{R} \cup \mathcal{H}}$ terminating. A rule in $\mathcal{H}$ plays the role of an *induction hypothesis* for proving that the equations in $\mathcal{E}$ are inductive theorems, and is called an *induction rule*.

[3] A more standard definition of this property would be that for every defined or calculation symbol f and suitable ground constructor terms $s_1, \ldots, s_n$, the term $f(s_1, \ldots, s_n)$ reduces. As observed in [14, Appendix A], this definition is equivalent.

SIMPLIFICATION If $s \approx t\,[\varphi] \to_{\mathcal{R}\cup\mathcal{H}} u \approx t\,[\psi]$, where $\approx$ is seen as a fresh constructor for the purpose of constrained term reduction,[4] then we may derive:

$$(\mathcal{E} \uplus \{(s \simeq t\ [\varphi])\}, \mathcal{H}) \ \vdash_{\mathtt{ri}}\ (\mathcal{E} \cup \{(u \approx t\ [\psi])\}, \mathcal{H})$$

DELETION If $s = t$ or φ is not satisfiable, we can delete $s \approx t\ [\varphi]$ from $\mathcal{E}$:

$$(\mathcal{E} \uplus \{s \approx t\ [\varphi]\}, \mathcal{H}) \ \vdash_{\mathtt{ri}}\ (\mathcal{E}, \mathcal{H})$$

EXPANSION Let $\mathit{Expd}(s, t, \varphi, p)$ be a set containing, for all rules $\ell \to r[\psi] \in \mathcal{R}$[5] such that ℓ is unifiable with $s_{|p}$ with most general unifier γ and $\varphi\gamma \wedge \psi\gamma$ is (or may be)[6] satisfiable, an equation $s' \approx t'\ [\varphi']$ where $s[\ell]_p\gamma \approx t\gamma\ [(\varphi\gamma) \wedge (\psi\gamma)] \to_{\mathcal{R}} s' \approx t'\ [\varphi']$ with rule $\ell \to r\ [\psi]$ at position $1 \cdot p$. Here, as in SIMPLIFICATION, $\approx$ is seen as a fresh constructor for the purpose of constrained term reduction. Intuitively, *Expd* generates all resulting equations if a ground constructor instance of $s \approx t\ [\varphi]$ is reduced at position p of s.
Now, if p is a position of s such that $s_{|p}$ is *basic* (i.e., $s_{|p} = f(s_1, \ldots, s_n)$ with f a defined symbol and all s_i constructor terms) we may derive:

$$(\mathcal{E} \uplus \{s \simeq t\ [\varphi]\}, \mathcal{H}) \ \vdash_{\mathtt{ri}}\ (\mathcal{E} \cup \mathit{Expd}(s, t, \varphi, p), \mathcal{H})$$

If, moreover, $\mathcal{R} \cup \mathcal{H} \cup \{s \to t\ [\varphi]\}$ is terminating, we may even derive:

$$(\mathcal{E} \uplus \{s \simeq t\ [\varphi]\}, \mathcal{H}) \ \vdash_{\mathtt{ri}}\ (\mathcal{E} \cup \mathit{Expd}(s, t, \varphi, p), \mathcal{H} \cup \{s \to t\ [\varphi]\})$$

Note that, if $\to_{\mathcal{R}}$ is non-deterministic (which may for instance happen when considering irregular rules), we can choose how to build *Expd*.

EQ-DELETION If all $s_i, t_i \in \mathcal{T}\mathit{erms}(\Sigma_{\mathit{theory}}, \mathit{Var}(\varphi))$, then we can derive:

$$(\mathcal{E} \uplus \{C[s_1, \ldots, s_n] \simeq C[t_1, \ldots, t_n]\ [\varphi]\}, \mathcal{H}) \ \vdash_{\mathtt{ri}}$$
$$(\mathcal{E} \cup \{C[s_1, \ldots, s_n] \approx C[t_1, \ldots, t_n]\ [\varphi \wedge \neg(\textstyle\bigwedge_{i=1}^{n} s_i = t_i)]\}, \mathcal{H})$$

$C[\,]$ is allowed to contain symbols in Σ_{theory}. Intuitively, if $\bigwedge_{i=1}^{n} s_i = t_i$ holds, then $C[s_1, \ldots, s_n]\gamma \leftrightarrow^*_{\mathcal{R}_{\mathtt{calc}}} C[t_1, \ldots, t_n]\gamma$ and thus, we are done. We exclude this case from the equation by adding $\neg(\bigwedge_{i=1}^{n} s_i = t_i)$ to the constraint.

GENERALIZATION If for all substitutions γ which respect φ there is a substitution δ which respects ψ with $s\gamma = s'\delta$ and $t\gamma = t'\delta$, then we can derive:

$$(\mathcal{E} \uplus \{s \approx t\ [\varphi]\}, \mathcal{H}) \ \vdash_{\mathtt{ri}}\ (\mathcal{E} \cup \{s' \approx t'\ [\psi]\}, \mathcal{H})$$

[4] It is not enough if $s\,[\varphi] \to_{\mathcal{R}} u\,[\psi]$: when reducing constrained terms, we may manipulate unused variables at will, which causes problems if they are used in t. For example, $f(x + 0)\,[x > y] \sim f(x + 0)\,[z = x + 0] \to_{\mathtt{base}} f(z)\,[z = x + 0] \sim f(x)\,[y < x]$, but we would not want to replace an equation $f(x+0) \approx g(y)\ [x > y]$ by $f(x) \approx g(y)\ [x < y]$!

[5] Here, we assume that the variables in the rules are distinct from the ones in s, t, φ.

[6] Although we do not *have* to include equations in $\mathit{Expd}(s, t, \varphi, p)$ which correspond to rules that give an unsatisfiable constraint, it is sometimes convenient to postpone the satisfiability check; the resulting equations can be removed with DELETION.

The first three of these rules originate in [18], but they are adapted in several ways. Partially, this is because we consider LCTRSs rather than plain TRSs, and have to handle the constraints: hence we use constrained reduction rather than normal reduction in SIMPLIFICATION, and include an unsatisfiability case in DELETION. In EXPANSION, we have made more structural changes; our definition also differs from the corresponding rules in [5,20], where the method is defined for different styles of constrained rewriting.

To start, we use constrained reduction, whereas the authors of [18,5,20] use direct instantiation (e.g. $Expd(s,t,p)$ contains elements $s[r\gamma]_p \approx t$ when $\ell \to r \in \mathcal{R}$ and $s_{|p}$ unifies with ℓ with most general unifier γ). This was changed to better handle irregular rules, especially those where the right-hand side introduces fresh variables, i.e. $\ell \to r\ [\varphi]$ where $Var(r) \cap Var(\varphi) \not\subseteq Var(\ell)$. Such rules occur for example in transformed iterative functions where variables are declared but not immediately initialized. The alternative formulation of $\mathcal{R}$ in Section 5, which is essential for our lemma generalization technique, also uses such irregular rules.

Second, the case where no rule is added is new. This is needed to allow progress in cases when adding the rule might cause loss of termination. It somewhat corresponds to, but is strictly stronger than, CASE-SIMPLIFY in [5].

EQ-DELETION originates in [20] and can, in combination with DELETION, be seen as a generalized variant of THEORY$_\top$ in [5]. Most importantly, this inference rule provides a link between the equation part $s \approx t$ and the constraint. The last rule, GENERALIZATION, can be seen as a special case of POSTULATE in [18]. By generalizing an equation, the EXPANSION rule gives more powerful induction rules, which (as discussed in Section 5) is often essential to prove a theorem.

The inference rules are used for *rewriting induction* by the following theorem:

Theorem 10. *Let an LCTRS with rules $\mathcal{R}$ and signature Σ, satisfying the restrictions from Section 4.1, be given; let $\mathcal{E}$ be a finite set of equations. If $(\mathcal{E},\emptyset) \vdash_{\mathtt{ri}} \cdots \vdash_{\mathtt{ri}} (\emptyset,\mathcal{H})$, then every $e \in \mathcal{E}$ is an inductive theorem of $\mathcal{R}$.*

Proof Sketch: We follow the proof method of [20] (with a few adaptations), which builds on the original proof idea in [18]. That is, we define $\leftrightarrow_{\mathcal{E}}$ in the expected way (treating an equation as a rule) and prove the equivalent statement that $\leftrightarrow^*_{\mathcal{E}} \subseteq \leftrightarrow^*_{\mathcal{R}}$ on ground terms by making the following observations:

1. If $(\mathcal{E}_1,\mathcal{H}_1) \vdash_{\mathtt{ri}} (\mathcal{E}_2,\mathcal{H}_2)$, then $\leftrightarrow_{\mathcal{E}_1} \subseteq \to^*_{\mathcal{R}\cup\mathcal{H}_2} \cdot (\leftrightarrow_{\mathcal{E}_2} \cup =) \cdot \leftarrow^*_{\mathcal{R}\cup\mathcal{H}_2}$ on ground terms (which we see by a careful analysis of all inference rules); using induction we obtain that $\leftrightarrow_{\mathcal{E}} \subseteq \to^*_{\mathcal{R}\cup\mathcal{H}} \cdot = \cdot \leftarrow^*_{\mathcal{R}\cup\mathcal{H}} \subseteq \leftrightarrow^*_{\mathcal{R}\cup\mathcal{H}}$.
2. If $(\mathcal{E}_1,\mathcal{H}_1) \vdash_{\mathtt{ri}} (\mathcal{E}_2,\mathcal{H}_2)$ and $\to_{\mathcal{R}\cup\mathcal{H}_1} \subseteq \to_{\mathcal{R}} \cdot \to^*_{\mathcal{R}\cup\mathcal{H}_1} \cdot (\leftrightarrow_{\mathcal{E}_1} \cup =) \cdot \leftarrow^*_{\mathcal{R}\cup\mathcal{H}_1}$, then also $\to_{\mathcal{R}\cup\mathcal{H}_2} \subseteq \to_{\mathcal{R}} \cdot \to^*_{\mathcal{R}\cup\mathcal{H}_2} \cdot (\leftrightarrow_{\mathcal{E}_2} \cup =) \cdot \leftarrow^*_{\mathcal{R}\cup\mathcal{H}_2}$ (which follows by a case analysis, paying particular attention to the EXPANSION rule); using induction we obtain that $\to_{\mathcal{R}\cup\mathcal{H}} \subseteq \to_{\mathcal{R}} \cdot \to^*_{\mathcal{R}\cup\mathcal{H}} \cdot \leftarrow^*_{\mathcal{R}\cup\mathcal{H}}$.
3. By point 2 and induction on $\to_{\mathcal{R}\cup\mathcal{H}}$, we find that $\leftrightarrow^*_{\mathcal{R}} = \leftrightarrow^*_{\mathcal{R}\cup\mathcal{H}}$.

Details are provided in our technical report [14]. □

Following e.g. [1], there are many other potential inference rules we could consider. For space reasons, we limit interest to the rules needed for our examples.

4.3 Some Illustrative Examples

To show how the method works, recall the sum1 and sum2 rules from Section 3 (page 7). We want to see that these two implementations are equivalent, at least when the input makes sense, so the given length is at least 0 and does not exceed the array size. This is the case if the following equation is an inductive theorem:

$$\text{(A)}\quad \mathsf{sum1}(a,k) \approx \mathsf{sum2}(a,k)\ [0 \le k \le \mathsf{size}(a)]$$

Thus, we start the procedure with $(\{\text{(A)}\},\emptyset)$. From Simplification, we obtain:

$$(\{\text{(B)}\quad \mathsf{u}(a,k,0,0) \approx \mathsf{sum2}(a,k)\ [0 \le k \le \mathsf{size}(a)]\},\emptyset)$$

None of Simplification, EQ-deletion, Deletion and Constructor is applicable, so we apply Expansion to the right-hand side of (B) at the root. Since $k \le \mathsf{size}(a)$ and $k-1 \ge \mathsf{size}(a)$ cannot both hold, the error rule leads to an unsatisfiable constraint. Therefore, this step only gives two new equations:

$$\left(\left\{\begin{array}{c}\text{(C)} : \mathsf{return}(0) \approx \mathsf{u}(a,k,0,0)\ [0 \le k \le \mathsf{size}(a) \wedge k \le 0] \\ \text{(D)} : \mathsf{w}(\mathsf{select}(a,k-1),\mathsf{sum2}(a,k-1)) \approx \mathsf{u}(a,k,0,0) \\ [0 \le k \le \mathsf{size}(a) \wedge 0 \le k-1 < \mathsf{size}(a)]\end{array}\right\}, \{\ (\mathrm{B}^{-1})\ \}\right)$$

Here, (B^{-1}) should be read as the rule generated from (B) right-to-left, so $\mathsf{sum2}(a,k) \to \mathsf{u}(a,k,0,0)\ [0 \le k \le \mathsf{size}(a)]$. We use Simplification with rule (4) to reduce (C) to $\mathsf{return}(0) \approx \mathsf{return}(0)\ [\ldots]$, which we quickly delete. Simplifying the right-hand side of (D) with rule (3), we obtain $(\{\text{(E)}\},\{(\mathrm{B}^{-1})\})$, with:

$$\begin{array}{c}\text{(E)} : \mathsf{w}(\mathsf{select}(a,k-1),\mathsf{sum2}(a,k-1)) \approx \mathsf{u}(a,k,0+\mathsf{select}(a,0),0+1) \\ [0 \le k \le \mathsf{size}(a) \wedge 0 \le k-1 < \mathsf{size}(a)]\end{array}$$

Next we use Simplification with the calculation rules. As these rules are irregular, this requires some care. There are three standard ways to do this:

- if $s \to_{\mathtt{calc}} t$ then $s\,[\varphi] \to_{\mathcal{R}} t\,[\varphi]$, e.g. $\mathsf{f}(0+1) \approx r\ [\varphi]$ reduces to $\mathsf{f}(1) \approx r\ [\varphi]$;
- a calculation can be replaced by a fresh variable, which is defined in the constraint, e.g. $\mathsf{f}(x+1) \approx r\ [\varphi]$ reduces to $\mathsf{f}(y) \approx r\ [\varphi \wedge y = x+1]$;
- a calculation *already* defined in the constraint can be replaced by the relevant variable, e.g. $\mathsf{f}(x+1) \approx r\ [\varphi \wedge y = x+1]$ reduces to $\mathsf{f}(y) \approx r\ [\varphi \wedge y = x+1]$.

These ways are not functionally different; if an equation e reduces both to e_1 and e_2 with a calculation at the same position, then it is easy to see that $e_1 \sim e_2$.

We can do more: recall that, by definition of constrained term reduction, we can rewrite a constraint φ with variables $\overrightarrow{x}, \overrightarrow{y}$ in a constrained term $s\,[\varphi]$, to any constraint ψ over $\overrightarrow{x}, \overrightarrow{z}$ such that $\exists\overrightarrow{y}[\varphi]$ is equivalent to $\exists\overrightarrow{z}[\psi]$ (if $Var(s) = \{\overrightarrow{x}\}$). We use this observation to write constraints in a simpler form after Simplification or Expansion, for instance by removing redundant clauses.

Using six more Simplification steps with the calculation rules on (E), and writing the constraint in a simpler form, we obtain:

$$\left(\left\{\begin{array}{l}\text{(F)} : \quad \mathsf{w}(n,\mathsf{sum2}(a,k')) \approx \mathsf{u}(a,k,r,1)\ \ [k' = k-1 \wedge \\ \qquad 0 \le k' < \mathsf{size}(a) \wedge n = \mathsf{select}(a,k') \wedge r = 0 + \mathsf{select}(a,0)]\end{array}\right\}, \{\ (\mathrm{B}^{-1})\ \}\right)$$

Then, using Simplification with the induction rule (B^{-1}):

$$\left(\left\{\begin{array}{ll}(\mathrm{G}): & \mathsf{w}(n,\mathsf{u}(a,k',0,0)) \approx \mathsf{u}(a,k,r,1) \;\; [k' = k-1 \wedge \\ & 0 \leq k' < \mathsf{size}(a) \wedge n = \mathsf{select}(a,k') \wedge r = 0 + \mathsf{select}(a,0)]\end{array}\right\}, \{\,(\mathrm{B}^{-1})\,\}\right)$$

As the simpler inference rules do not apply, we expand in the right-hand side:

$$\left(\left\{\begin{array}{ll}(\mathrm{H}): & \mathsf{u}(a,k,r+\mathsf{select}(a,1),1+1) \approx \mathsf{w}(n,\mathsf{u}(a,k',0,0)) \\ & [k' = k-1 \wedge 0 \leq k' < \mathsf{size}(a) \wedge n = \mathsf{select}(a,k') \wedge \\ & \quad r = 0 + \mathsf{select}(a,0) \wedge 1 < k \wedge 0 \leq 1 < \mathsf{size}(a)] \\ (\mathrm{I}): & \mathsf{return}(r) \approx \mathsf{w}(n,\mathsf{u}(a,k',0,0)) \; [k' = k-1 \wedge 0 \leq k' < \\ & \mathsf{size}(a) \wedge n = \mathsf{select}(a,k') \wedge r = 0 + \mathsf{select}(a,0) \wedge \; 1 \geq k]\end{array}\right\}, \left\{\begin{array}{l}(\mathrm{B}^{-1}) \\ (\mathrm{G}^{-1})\end{array}\right\}\right)$$

We have again omitted the error rule, as the corresponding constraint is not satisfiable. For (I), the constraint implies that $k = 1$, so Simplification with rule (4) followed by (9) and prettifying the constraint gives $\mathsf{return}(r) \approx \mathsf{return}(n+0)$ $[k' = 0 < \mathsf{size}(a) \wedge n = \mathsf{select}(a,k') \wedge r = \mathsf{select}(a,0)]$. Eq-deletion gives an unsatisfiable constraint $\ldots \wedge \neg(r = n + 0)$; we complete with Deletion.

We continue with $(\{(\mathrm{H})\},\{(\mathrm{B}^{-1}),(\mathrm{G}^{-1})\})$. After applying Simplification with (3) and calculation rules a few times, we have $(\{(\mathrm{J})\}, \{(\mathrm{B}^{-1}),(\mathrm{G}^{-1})\})$:

$$\begin{array}{rl}(\mathrm{J}): \mathsf{u}(a,k,r_1,2) \approx \mathsf{w}(n,\mathsf{u}(a,k',r,1)) & [k' = k-1 \wedge 0 \leq k' \wedge 1 < k \leq \mathsf{size}(a) \wedge \\ & n = \mathsf{select}(a,k') \wedge r = 0 + \mathsf{select}(a,0) \wedge r_1 = r + \mathsf{select}(a,1)]\end{array}$$

Here, we have used the third style of calculation simplification to reuse r.

We can use Expansion again, this time on the left-hand side. But now a pattern starts to arise. If we continue like this, simplifying as long as we can, and then using whichever of the other core rules is applicable, we get:

$$\begin{array}{l}(\mathrm{K}): \mathsf{u}(a,k,r_2,3) \approx \mathsf{w}(n,\mathsf{u}(a,k',r_1,2)) \; [k' = k-1 \wedge 2 < k \leq \mathsf{size}(a) \wedge \ldots] \\ (\mathrm{L}): \mathsf{u}(a,k,r_3,4) \approx \mathsf{w}(n,\mathsf{u}(a,k',r_2,3)) \; [k' = k-1 \wedge 3 < k \leq \mathsf{size}(a) \wedge \ldots]\end{array}$$

That is, we have a *divergence*: a sequence of increasingly complex equations, each generated from the same leg in an Expansion (see also the *divergence critic* in [22]). Yet the previous induction rules never apply to the new equation.

So, consider the following equation (we will say more about it in Section 5):

$$\begin{array}{rl}(\mathrm{M}): \mathsf{u}(a,k,r,i) \approx \mathsf{w}(n,\mathsf{u}(a,k',r',i')) & [k' = k-1 \wedge 0 \leq i' < k \leq \mathsf{size}(a) \wedge \\ & i' = i-1 \wedge r = r' + \mathsf{select}(a,i') \wedge n = \mathsf{select}(a,k')]\end{array}$$

It is easy to see that (J) is an instance of (M); we apply Generalization and continue with $(\{(\mathrm{M})\}, \{(\mathrm{B}^{-1}),(\mathrm{G}^{-1})\})$. Using Expansion, we obtain:

$$\left(\left\{\begin{array}{ll}(\mathrm{N}): & \mathsf{u}(a,k,r+\mathsf{select}(a,i),i+1) \approx \mathsf{w}(n,\mathsf{u}(a,k',r',i')) \\ & [k' = k-1 \wedge 0 \leq i' < k \leq \mathsf{size}(a) \wedge i' = i-1 \wedge r = r' + \\ & \mathsf{select}(a,i') \wedge n = \mathsf{select}(a,k') \;\wedge i < k \;\wedge 0 \leq i < \mathsf{size}(a)] \\ (\mathrm{O}): & \mathsf{return}(r) \approx \mathsf{w}(n,\mathsf{u}(a,k',r',i')) \\ & [k' = k-1 \wedge 0 \leq i' < k \leq \mathsf{size}(a) \wedge i' = i-1 \wedge \\ & r = r' + \mathsf{select}(a,i') \wedge n = \mathsf{select}(a,k') \wedge i \geq k]\end{array}\right\}, \left\{\begin{array}{c}(\mathrm{B}^{-1}) \\ (\mathrm{G}^{-1}) \\ (\mathrm{M})\end{array}\right\}\right)$$

Again, the result of the error rule is omitted, because $i < 0$ cannot hold if both $0 \leq i'$ and $i' = i - 1$, and $i \geq \mathsf{size}(a)$ cannot hold if both $i < k$ and $k \leq \mathsf{size}(a)$.

Consider (O). Investigating the constraint, we can simplify it with rules (4) and (9), and then complete with EQ-DELETION and DELETION.

Only (N) remains. We simplify this equation with the normal rules, giving:

$$\mathsf{u}(a, k, r'', i'') \approx \mathsf{w}(n, \mathsf{u}(a, k', r, i))$$
$$[k' = k - 1 \wedge 0 \leq i' < k \leq \mathsf{size}(a) \wedge i' = i - 1 \wedge r = r' + \mathsf{select}(a, i') \wedge$$
$$n = \mathsf{select}(a, k') \wedge i < k \wedge 0 \leq i < \mathsf{size}(a) \wedge i'' = i + 1 \wedge r'' = r + \mathsf{select}(a, i)]$$

But now note that the induction rule (M) applies! This rule is irregular, so for the constrained reduction step we use a substitution that also affects variables not occurring in its left-hand side: $\gamma = [a := a, k := k, r := r'', i := i'', n := n, k' := k', r' := r, i' := i]$. Using SIMPLIFICATION, the equation is reduced to $\mathsf{w}(n, \mathsf{u}(a, k', r, i)) \approx \mathsf{w}(n, \mathsf{u}(a, k', r, i))$ $[\ldots]$, which is removed using DELETION.

As $(\{(\mathrm{A})\},\emptyset) \vdash_{\mathtt{ri}}^{*} (\emptyset, \mathcal{H})$ for some $\mathcal{H}$, we see that (A) is an inductive theorem.

For another example, let us look at an assignment to implement `strlen`, a string function which operates on 0-terminated `char` arrays. As `char` is a numeric data type, the LCTRS translation can implement this as integer arrays again (although using another underlying sort $\mathcal{I}_{\mathsf{char}}$ would make little difference).

The example function and its LCTRS translation are as follows:

```
int strlen(char *str) {
  for (int i = 0; ; i++)
    if (str[i] == 0) return i;
}
```

$$\begin{array}{lll}
(10) & \mathsf{strlen}(x) \to \mathsf{u}(x, 0) & \\
(11) & \mathsf{u}(x, i) \to \mathsf{error} & [i < 0 \vee i \geq \mathsf{size}(x)] \\
(12) & \mathsf{u}(x, i) \to \mathsf{return}(i) & [0 \leq i < \mathsf{size}(x) \wedge \mathsf{select}(x, i) = 0] \\
(13) & \mathsf{u}(x, i) \to \mathsf{u}(x, i + 1) & [0 \leq i < \mathsf{size}(x) \wedge \mathsf{select}(x, i) \neq 0]
\end{array}$$

Note that the overflow checks guarantee termination.

To see that strlen does what we would expect it to do, we want to know that for *valid C-strings*, $\mathsf{strlen}(a)$ returns the first integer i such that $a[i] = 0$:

$$\text{(P)} \quad \mathsf{strlen}(x) \approx \mathsf{return}(n)$$
$$[0 \leq n < \mathsf{size}(x) \wedge \forall i \in \{0, n - 1\}[\mathsf{select}(x, i) \neq 0] \wedge \mathsf{select}(x, n) = 0]$$

Here, we use bounded quantification, which, as described in Section 2, can be seen as syntactic sugar for an additional predicate, e.g. nonzero_until.

Starting with $(\{(\mathrm{P})\},\emptyset)$, we first use SIMPLIFICATION with rule (10), creating:

$$\text{(Q)} \quad \mathsf{u}(x, 0) \approx \mathsf{return}(n)$$
$$[0 \leq n < \mathsf{size}(x) \wedge \forall i \in \{0, n - 1\}[\mathsf{select}(x, i) \neq 0] \wedge \mathsf{select}(x, n) = 0]$$

We continue with Expansion; since the constraint implies that $0 < \mathsf{size}(x)$, the error case (11) gives an unsatisfiable constraint; we only get two new equations:

$$\begin{aligned}
\text{(R)}\quad \mathsf{return}(0) \approx \mathsf{return}(n)\ [&0 \le n < \mathsf{size}(x) \wedge \forall i \in \{0, n-1\}[\mathsf{select}(x,i) \neq 0] \wedge \\
&\mathsf{select}(x,n) = 0 \wedge 0 \le 0 < \mathsf{size}(x) \wedge \mathsf{select}(x,0) = 0] \\
\text{(S)}\quad \mathsf{u}(x, 0+1) \approx \mathsf{return}(n)\ [&0 \le n < \mathsf{size}(x) \wedge \forall i \in \{0, n-1\}[\mathsf{select}(x,i) \neq 0] \wedge \\
&\mathsf{select}(x,n) = 0 \wedge 0 \le 0 < \mathsf{size}(x) \wedge \mathsf{select}(x,0) \neq 0]
\end{aligned}$$

As the constraint of (R) implies that $n = 0$ (because of the quantification and $\mathsf{select}(x,0) = 0$), we can remove (R) using EQ-deletion and Deletion.

As for (S), we simplify with a calculation, and expand again. This gives an equation $\mathsf{return}(1) \approx \mathsf{return}(n)$ [...] that we can quickly remove again, and an equation (T) which is simplified, expanded and eq-deleted/deleted into:

$$\begin{aligned}
\text{(U)}\ \mathsf{u}(x, 2+1) \approx \mathsf{return}(n)\ [&0 \le n < \mathsf{size}(x) \wedge \forall i \in \{0, n-1\}[\mathsf{select}(x,i) \neq 0] \wedge \\
&\mathsf{select}(x,n) = 0 \wedge 0 < \mathsf{size}(x) \wedge \mathsf{select}(x,0) \neq 0 \wedge \\
&1 < \mathsf{size}(x) \wedge \mathsf{select}(x,1) \neq 0 \wedge 2 < \mathsf{size}(x) \wedge \mathsf{select}(x,2) \neq 0]
\end{aligned}$$

Simplifying and reformulating the constraint, we obtain:

$$\begin{aligned}
\text{(V)}\ \mathsf{u}(x,3) \approx \mathsf{return}(n)\ [&0 \le n < \mathsf{size}(x) \wedge \forall i \in \{0, n-1\}[\mathsf{select}(x,i) \neq 0] \wedge \\
&\mathsf{select}(x,n) = 0 \wedge 0 \le 2 < \mathsf{size}(x) \wedge \forall j \in \{0,2\}[\mathsf{select}(x,j) \neq 0]]
\end{aligned}$$

Note that we grouped together the $\neq 0$ statements into a quantification, which looks a lot like the other quantification in the constraint. We apply Generalization to obtain $(\{\text{(W)}\}, \{\dots\})$, where (W) is $\mathsf{u}(x,k) \approx \mathsf{return}(n)$ $[\varphi]$ with:

$$\begin{aligned}
\varphi : [&k = m+1 \wedge 0 \le n < \mathsf{size}(x) \wedge \forall i \in \{0, n-1\}[\mathsf{select}(x,i) \neq 0] \wedge \\
&\mathsf{select}(x,n) = 0 \wedge 0 \le m < \mathsf{size}(x) \wedge \forall j \in \{0,m\}[\mathsf{select}(x,j) \neq 0]
\end{aligned}$$

Obviously, (V) is an instance of (W); we proceed with Expansion on (W) to obtain the proof status $(\{\text{(X)}, \text{(Y)}, \text{(Z)}\}, \{\dots, \text{(W)}\})$, where:

$$\begin{aligned}
\text{(X)}\quad & \mathsf{error} \approx \mathsf{return}(n)\ [\varphi \wedge (k < 0 \vee k \ge \mathsf{size}(x))] \\
\text{(Y)}\quad & \mathsf{return}(k) \approx \mathsf{return}(n)\ [\varphi \wedge 0 \le k < \mathsf{size}(x) \wedge \mathsf{select}(x,k) = 0] \\
\text{(Z)}\quad & \mathsf{u}(x, k+1) \approx \mathsf{return}(n)\ [\varphi \wedge 0 \le k < \mathsf{size}(x) \wedge \mathsf{select}(x,k) \neq 0]
\end{aligned}$$

For all cases, note that the two $\forall$ statements, together with $\mathsf{select}(x,n) = 0$, imply that $m < n$, so $k \le n$. Hence the constraint of (X) is unsatisfiable: $k = m+1$ and $0 \le m$ imply that $k \not< 0$, and $k \le n,\ k \ge \mathsf{size}(x)$ imply that $n \not< \mathsf{size}(x)$. By Deletion, we remove (X). For (Y), we use EQ-deletion. Note that the two $\forall$ statements, together with $\mathsf{select}(x,k) = 0$, imply that $n - 1 < k$, so $n \le k$. Since also $k \le n$, the resulting constraint is unsatisfiable; we use Deletion again.

Finally, simplifying (Z) with a calculation, and reformulating the constraint:

$$\begin{aligned}
&\mathsf{u}(x,p) \approx \mathsf{return}(n) \\
[&p = k+1 \wedge \mathsf{select}(x,n) = 0 \wedge 0 \le n < \mathsf{size}(x) \wedge \forall i \in \{0, n-1\}[\mathsf{select}(x,i) \neq 0] \\
&\wedge\ 0 \le k < \mathsf{size}(x) \wedge \forall j \in \{0,k\}[\mathsf{select}(x,j) \neq 0]\ \wedge \text{some constraints on } m]
\end{aligned}$$

The induction rule (W) lets us simplify this to $\mathsf{return}(n) \approx \mathsf{return}(n)$ [...], which is easily removed using Deletion.

5 Lemma Generalization by Dropping Initializations

Divergence, like we encountered in both examples of Section 4, is very common in inductive theorem proving. This is only natural: in mathematical proofs, when basic induction fails to prove a theorem, we often need a more general claim to obtain a stronger induction hypothesis. Viewed in this light, the generalization of equations, or the generation of suitable auxiliary lemmas is not only part, but even at the heart, of inductive theorem proving. Consequently, this subject has been extensively investigated [3,10,11,16,21,22]. Candidates for such equations are typically generated during solving, when the proof state is in divergence.

In this section, we propose a new method, specialized for constrained systems. The generalizations from Section 4 were found using this technique. Although the method is very simple (at its core, we just drop a part of the constraint), it is particularly effective for LCTRSs obtained from procedural programs.

First, let us state the rules of our sum example differently. When the right-hand side of a rule has a subterm $f(\dots, n, \dots)$ with f defined and n a value, we replace n by a fresh variable v_i, and add $v_i = n$ to the constraint. In the LCTRS $\mathcal{R}_{sum}$ from page 7, rules (2)–(9) are not changed, but (1) is replaced by:

$$(1')\quad \mathsf{sum1}(arr, n) \to \mathsf{u}(arr, n, v_1, v_2)\quad [v_1 = 0 \wedge v_2 = 0]$$

Evidently, these altered rules generate the same rewrite relation as the original.

Consider what happens now if we use the same steps as in Section 4.3. We do not rename the variables v_i in Expansion, and ignore the $v_i = n$ clauses when simplifying the presentation of a constrained term. The resulting induction has the same shape, but with more complex equations. Some instances:

$$\begin{aligned}
(\mathrm{B}'):&\quad \mathsf{u}(a,k,v_1,v_2) \approx \mathsf{sum2}(a,k)\ [0 \le k \le \mathsf{size}(a) \wedge v_1 = 0 \wedge v_2 = 0]\\
(\mathrm{F}'):&\quad \mathsf{w}(n, \mathsf{sum2}(a,k')) \approx \mathsf{u}(a,k,r_0,i_0)\\
&\quad [k' = k-1 \wedge 0 \le k' < \mathsf{size}(a) \wedge v_1 = 0 \wedge v_2 = 0 \wedge n = \mathsf{select}(a,k') \wedge\\
&\qquad r_0 = v_1 + \mathsf{select}(a, v_2) \wedge i_0 = v_2 + 1]\\
(\mathrm{J}'):&\quad \mathsf{u}(a,k,r_1,i_1) \approx \mathsf{w}(n, \mathsf{u}(a,k',r_0,i_0))\\
&\quad [k' = k-1 \wedge 0 \le k' < \mathsf{size}(a) \wedge v_1 = 0 \wedge v_2 = 0 \wedge n = \mathsf{select}(a,k') \wedge\\
&\qquad r_0 = v_1 + \mathsf{select}(a, v_2) \wedge i_0 = v_2 + 1 \wedge i_0 < k \wedge 0 \le i_0 < \mathsf{size}(a) \wedge\\
&\qquad i_1 = i_0 + 1 \wedge r_1 = r_0 + \mathsf{select}(a, i_0)]
\end{aligned}$$

Continuing from (J′), we get equations $\mathsf{u}(a,k,r_2,i_2) \approx \mathsf{w}(n, \mathsf{u}(a,k',r_1,i_1))\ [\varphi]$ and $\mathsf{u}(a,k,r_3,i_3) \approx \mathsf{w}(n, \mathsf{u}(a,k',r_2,i_2))\ [\psi]$ whose main part is the same as that of (J′), modulo renaming of variables, while the constraint grows. Essentially, we keep track of parts of the history of an equation in its constraint.

We generalize (J′) by dropping all clauses $v_i = q_i$, where v_i is an initialization variable. Remaining occurrences of v_i are renamed to avoid confusion. This gives:

$$\begin{aligned}
(\mathrm{M}')\ \mathsf{u}(a,k,r_1,i_1) \approx \mathsf{w}(n, \mathsf{u}(a,k',r_0,i_0))\quad &[k' = k-1 \wedge 0 \le k' < \mathsf{size}(a) \wedge\\
n = \mathsf{select}(a,k') \wedge r_0 = x_1 + \mathsf{select}(a,x_2) \wedge i_0 = x_2 + 1 \wedge i_0 < k \wedge&\\
0 \le i_0 < \mathsf{size}(a) \wedge i_1 = i_0 + 1 \wedge r_1 = r_0 + \mathsf{select}(a, i_0)]&
\end{aligned}$$

Note that (M′) ∼ (M): the clauses with x_1 and x_2 can be removed, as suitable x_1, x_2 always exist. Continuing with (M′) completes the proof as before.

Discussion. Thus, our equation generalization technique is very straightforward to use: we merely replace initializations by variables in the original rules, then remove the definitions of those initializations when a divergence is detected.

The only downside is that, in order to use this technique, we have to use the altered rules from the beginning, so we keep track of the v_i variables throughout the recursive procedure. For an automatic analysis this is no problem, however.

Note that we can only use this method if the equation part of the divergence has the same shape every time. This holds for sum, because the rule that causes the divergence has the form $\mathsf{u}(x_1, \ldots, x_n) \to \mathsf{u}(r_1, \ldots, r_n)\ [\varphi]$, preserving its outer shape. In general, the generalization method is most likely to be successful when analyzing tail-recursive functions (with accumulators), such as those obtained from procedural programs. This includes mutually recursive functions, like $\mathsf{u}(x_1, \ldots, x_n) \to \mathsf{w}(r_1, \ldots, r_m)\ [\varphi]$ and $\mathsf{w}(y_1, \ldots, y_m) \to \mathsf{u}(q_1, \ldots, q_n)\ [\psi]$. To analyze systems with general recursion, however, we will need different techniques.

The given generalization method also works for strlen from Section 4.3, and for strcpy. In these cases, we additionally have to collect multiple clauses into a quantified clause before generalizing, as was done for equation (W) in Section 4.3.

6 Implementation

We have implemented the rewriting induction and generalization methods in this paper in Ctrl, our tool for analyzing constrained term rewriting. As prerequisites, we have also implemented basic techniques to prove termination and quasi-reductivity. To deal with constraints, the tool is coupled both with a small internal reasoner and the (quantifier-capable) external SMT-solver Z3 [15].

The internal reasoner has two functions. First, it uses standard tricks to detect satisfiability or validity of simple statements, without a call to the external solver; this is both faster, and lets us optimize for often recurring questions (e.g. "find $n_1, \ldots, n_k \in \{-2, \ldots, 2\}$ such that φ is valid", as used for termination). Second, it simplifies the constraints of equations, for instance combining statements into quantifications. In addition, our notion of arrays is not supported by mainstream SMT-solvers, so we translate our array formulas into the SMT-LIB array-format; an array is encoded as a function from $\mathbb{Z}$ to $\mathbb{Z}$, with an additional variable encoding its size.

To obtain Ctrl, see: `http://cl-informatik.uibk.ac.at/software/ctrl/`.

Strategy. The rewriting induction method of Ctrl uses a simple strategy: we try, in the following order: EQ-DELETION and DELETION together, SIMPLIFICATION, EXPANSION, and GENERALIZATION (simply removing all $v_i = t$ definitions). When a rule succeeds, we continue from the start of the list. When we encounter an obviously unsolvable problem, or have gone too deep without removing any of the main equations, we backtrack and try something else. At the moment, divergence is not automatically detected, although this is an obvious extension.

To rewrite an equation in SIMPLIFICATION (and EXPANSION) with an irregular rule, we instantiate as many variables in the rule by existing variables as possible (as done for (N) in Section 4.3). Other variables are instantiated with fresh

variables. When simplifying constraints, clauses which are clearly implied by other clauses (ignoring the $v_i = n$ definitions) or do not play a role are removed. Most importantly, Ctrl introduces *ranged quantifications* $\forall x \in \{k_1, \ldots, k_n\}[\varphi(x)]$ whenever possible (as we also saw in Section 4.3), provided $n \geq 3$. If a boundary of the range is a special variable v_i, we replace it by the value it is defined as, since it is typically better not to generalize the starting point of a quantification.

Experiments. To test performance of Ctrl, we used assignments from a group of students in the first-year programming course in Nagoya. Unfortunately, although we know how to translate C-programs to LCTRSs, we do not yet have an implementation. Therefore, we translated five groups by hand: `sum` (given n, implement $\sum_{i=1}^{n} i$), `fib` (calculate the first n Fibonacci numbers), `sumfrom` (given n and m, implement $\sum_{i=n}^{m} i$), `strlen` and `strcpy`. Due to the large effort of manually translating, we only use this small sample space. We considered two further assignments, with our own implementations: `arrsum` (the array summation from Section 4.3) and `fact` (the factorial function from Examples 1 and 7).

We quickly found that many implementations were incorrect: students had often forgotten to account for, e.g., negative input. Correcting for this (by altering the constraint, or excluding the benchmark), Ctrl automatically verified most queries, as summarized to the right. Here, for instance "3 / 5" means that 3 out of the 5 different correct functions could automatically be verified. The runtime includes only queries where Ctrl succeeded.

function	verified	time
`sum`	9 / 13	4.8
`fib`	10 / 12	11.4
`strlen`	3 / 5	16.2
`strcpy`	3 / 6	30.0
`sumfrom`	2 / 5	5.6
`arrsum`	1 / 1	14.2
`fact`	1 / 1	4.3

Investigating the failures, the main problem is termination. As Ctrl's termination module is not very strong yet, several times the initial LCTRS could not be handled; also, sometimes a natural induction rule was not introduced because it would cause non-termination (although in most of these cases, expanding at a different position still led to a proof). Another weakness is that sometimes, generalizing removes the relation between two variables (e.g. both x and y are initialized to 0, and are both increased by 1 in every loop iteration). This suggests a natural direction for improvements to the technique.

An evaluation page, including exact problem statements, is given at:
`http://cl-informatik.uibk.ac.at/software/ctrl/aplas14/`.

7 Related Work

The related work can be split into two categories. First, the literature on rewriting induction; and second, the work on program verification.

Rewriting Induction. Building on a long literature about rewriting induction (see e.g. [1,5,18,20]), the method for inductive theorem proving in this paper is primarily an adaptation of existing techniques to the new LCTRS formalism.

The most relevant related works are [5,20], where rewriting induction is defined for different styles of constrained rewriting. In both cases, the formalisms used are restricted to *integer* functions and predicates; it is not clear how they can be generalized to handle more advanced theories. LCTRSs offer a more gen-

eral setting, which allows us to use rewriting induction also for systems with for instance arrays, bitvectors or real numbers. Additionally, by not restricting the predicates in Σ_{theory}, we can handle (a limited form of) quantifiers in constraints.

To enable these advantages, we had to make subtle changes to the inference rules, in particular SIMPLIFICATION and EXPANSION. Our changes make it possible to modify constraints of an equation, and to handle *irregular* rules, where the constraint introduces fresh variables. This has the additional advantage that it enables EXPANSION steps when this would create otherwise infeasible rules.

Furthermore, the method requires a very different implementation from previous definitions: we need separate strategies to simplify constraints (e.g. deriving quantified statements), and, in order to permit the desired generality, must rely primarily on external solvers to manipulate constraints.

In addition to the adaptation of rewriting induction, we introduced a completely new lemma generalization technique, which offers a powerful tool for analyzing loops in particular. A similar idea (abstract the initialization values) is used in [16], but the execution is very different. In [16], an equation $s \approx t\ [\varphi]$ is generalized by first adapting $s \approx t$ using templates obtained from the rules, then generalizing φ using a set of relations between positions, which the proof process tracks. In our method, the constraint carries all information. Our method succeeds on all examples in [16], and on some where [16] fails (cf. [14, Appendix B]).

For *unconstrained* systems, there are several generalization methods in the literature, e.g., [10,11,21]. Mostly, these approaches are very different from ours. Most similar, perhaps, is [10], which also proposes a method to generalize initial values. As observed in [16], this is not sufficient even for our simplest benchmarks sum and fact since the argument for the loop variable cannot be generalized. In contrast, our method has no problem with such variables.

As far as we are aware, there is no other work for lemma generation of rewrite systems (or functional programs) obtained from procedural programs.

Automatic Program Verification. Although this paper is a primarily theoretical contribution to the field of constraint rewriting induction, our intended goal is to (automatically) verify correctness properties of procedural programs.

As mentioned in the introduction, however, most existing verifiers require human interaction. Exceptions are the fully automated tools in the *Competition on Software Verification* (SV-COMP, `http://sv-comp.sosy-lab.org/`), which verify program properties like reachability, termination and memory-safety.

However, comparing our approach to these tools does not seem useful. While we can, to some extent, tackle termination and memory-safety, the main topic of this paper is *equivalence*, which is not studied in SV-COMP. And while technically equivalence problems can be formulated as reachability queries (e.g., $f(x) \approx g(x)\ [c]$ is handled by the `main` function to the right), neither of the top two tools in the "recursive" category of SV-COMP halts succesfully (in two hours) for our simplest (integer) example sum.

```
int main() {
  int x =
   __VERIFIER_nondet_int();
  if (c && f(x) != g(x)) {
    ERROR: goto ERROR;
  }
  return 0;
}
```

8 Conclusions

In this paper,we have extended rewriting induction to the setting of LCTRSs. Furthermore, we have shown how this method can be used to prove correctness of procedural programs. LCTRSs seem to be a good analysis backend for this since the techniques from standard rewriting can typically be extended, and native support for logical conditions and data types like integers and arrays is present.

We have also introduced a new technique to generalize equations. The idea of this method is to identify constants used as *variable initializations*, keep track of them during the proof process, and abstract from these constants when a proof attempt diverges. The LCTRS setting is instrumental in the simplicity of this method, as it boils down to dropping a (cleverly chosen) part of a constraint.

In addition to the theory of these techniques, we provide an implementation that automatically verifies inductive theorems. Initial results on a small database of student programs are very promising. In future work, we will aim to increase the strength of this implementation and couple it with an automatic transformation tool which converts procedural programs into LCTRSs.

Acknowledgements. We are grateful to Stephan Falke, who contributed to an older version of this paper, and to both the IJCAR'14 and APLAS'14 referees for their helpful remarks.

References

1. Bouhoula, A.: Automated theorem proving by test set induction. Journal of Symbolic Computation, vol. 23(1), pp. 47–77. Elsevier (1997)
2. Bundy, A.: The automation of proof by mathematical induction. In: Voronkov, A., Robinson, A. (eds.) Handbook of Automated Reasoning, pp. 845–911. Elsevier (2001)
3. Bundy, A., Basin, D., Hutter, D., Ireland, A.: Rippling: Meta-Level Guidance for Mathematical Reasoning. Cambridge University Press (2005)
4. Falke, S., Kapur, D.: A term rewriting approach to the automated termination analysis of imperative programs. In: Schmidt. R.A. (ed.) Automated Deduction (CADE). LNCS, vol. 5663, pp. 277–293. Springer, Heidelberg (2009)
5. Falke, S., Kapur, D.: Rewriting induction + linear arithmetic = decision procedure. In: Gramlich, B., Miller, D., Sattler, U. (eds.) Automated Reasoning (IJCAR). LNAI, vol. 7364, pp. 241–255. Springer, Heidelberg (2012)
6. Falke, S., Kapur, D., Sinz, C.: Termination analysis of C programs using compiler intermediate languages. In: Schmidt-Schauß, M. (ed.) 22nd International Conference on Rewriting Techniques and Applications (RTA). LIPIcs, vol. 10, pp. 41–50. Dagstuhl, Leibniz (2011)
7. Falke, S.: Term Rewriting with Built-In Numbers and Collection Data Structures. Ph.D. thesis, University of New Mexico, Albuquerque, NM, USA (2009)
8. Furuichi, Y., Nishida, N., Sakai, M., Kusakari, K., Sakabe, T.: Approach to procedural-program verification based on implicit induction of constrained term rewriting systems. In: IPSJ Transactions on Programming, vol. 1(2), pp. 100–121 (2008), in Japanese. (**)

9. Huth, M., Ryan, M.: Logic in Computer Science: Modelling and Reasoning about Systems. Cambridge University Press (2000)
10. Kapur, D., Sakhanenko, N.A.: Automatic generation of generalization lemmas for proving properties of tail-recursive definitions. In: Basin, D., Wolff, B. (eds.) Theorem Proving in Higher Order Logics (TPHOLs). LNCS, vol. 2758, pp. 136–154. Springer, Heidelberg (2003)
11. Kapur, D., Subramaniam, M.: Lemma discovery in automated induction. In: McRobbie, M.A., Slaney, J.K. (eds.) Automated Deduction (CADE). LNCS, vol. 1104, pp. 538–552. Springer, Heidelberg (1996)
12. Kop, C.: Termination of LCTRSs. In: 13th International Workshop on Termination (WST), pp. 59–63 (2013)
13. Kop, C., Nishida, N.: Term rewriting with logical constraints. In: Fontaine, P. and Ringeissen, C. and Schmidt, R.A. (eds.) Frontiers of Combining Systems (FroCoS). LNCS, vol. 8152, pp. 343–358. Springer, Heidelberg (2013)
14. Kop, C., Nishida, N.: Towards verifying procedural programs using constrained rewriting induction. Technical report, University of Innsbruck (2014), `http://arxiv.org/abs/1409.0166`
15. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.H., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
16. Nakabayashi, N., Nishida, N., Kusakari, K., Sakabe, T., Sakai, M.: Lemma generation method in rewriting induction for constrained term rewriting systems. Computer Software, vol. 28(1), pp. 173–189 (2010), in Japanese. (**)
17. Otto, C., Brockschmidt, M., von Essen, C., Giesl, J.: Automated termination analysis of Java bytecode by term rewriting. In: Lynch, C. (ed.) 21st International Conference on Rewriting Techniques and Applications (RTA). LIPIcs, vol. 6, pp. 259–276. Dagstuhl, Leibniz (2010)
18. Reddy, U.S.: Term rewriting induction. In: Stickel, M.E. (ed.) 10th International Conference on Automated Deduction (CADE). LNCS, vol. 449, pp. 162–177. Springer, Heidelberg (1990)
19. Sakata, T., Nishida, N., Sakabe, T.: On proving termination of constrained term rewrite systems by eliminating edges from dependency graphs. In: Kuchen, H. (ed.) Functional and Constraint Logic Programming (WFLP). LNCS, vol. 6816, pp. 138–155. Springer, Heidelberg (2011)
20. Sakata, T., Nishida, N., Sakabe, T., Sakai, M., Kusakari, K.: Rewriting induction for constrained term rewriting systems. In: IPSJ Transactions on Programming, vol. 2(2), pp. 80–96 (2009), in Japanese. (**)
21. Urso, P., Kounalis, E.: Sound generalizations in mathematical induction. In: Theoretical Computer Science, vol. 323(1-3), pp. 443–471. Elsevier (2004)
22. Walsh, T.: A divergence critic for inductive proof. In: Journal of Artificial Intelligence Research, vol. 4, pp. 209–235. (1996)

(**) Translations or summaries of marked Japanese papers are available at:
`http://www.trs.cm.is.nagoya-u.ac.jp/crisys/`